

Desenvolvimento de Serviços e APIs

Centro Universitário UniSenac – Campus Pelotas

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Prof. Edécio Fernando Iepsen

Campos Calculados em APIs

► Usando `.map()` para transformar dados



O que são Campos Calculados?

- ▶ Campos **calculados** não existem originalmente no banco de dados.
- ▶ São **gerados dinamicamente** a partir dos dados existentes.
- ▶ Calculados dinamicamente no **backend**, antes de retornar a resposta
 - Exemplos:
 - nomeCompleto = nome + sobrenome
 - idade = anoAtual - anoNascimento
 - precoComDesconto = preco * 0.9



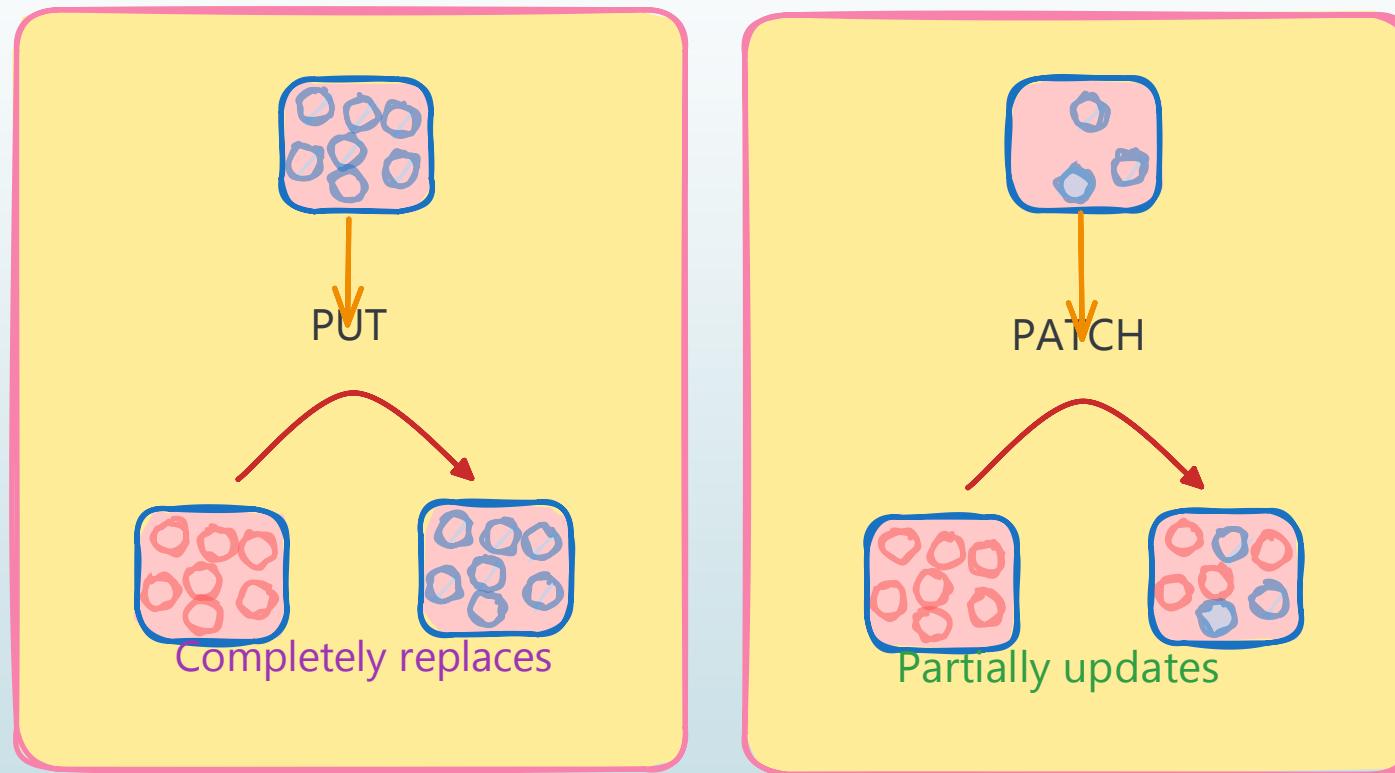
Por que Usar Campos Calculados?

- ➔ Personalizar a resposta da API.
- ➔ Evitar duplicação de lógica no frontend.
- ➔ Melhorar legibilidade dos dados.
- ➔ Manter o modelo de dados simples.

Usando `.map()` para criar campo calculado

```
const usuariosCompletos = usuarios.map(usuario => {  
  return {  
    ...usuario,  
    nomeCompleto: `${usuario.nome} ${usuario.sobrenome}`  
  };  
});
```

Alterações com PUT e PATCH



Put x Patch



→ C https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods/PATCH

mdn web docs References Guides Plus Curriculum Blog Tools NEW

Tecnologia Web para desenvolvedores > HTTP > Métodos de requisição HTTP > PATCH

Filter

CONNECT
DELETE
GET
HEAD
OPTIONS
PATCH
POST
PUT

PATCH

O método de requisição HTTP PATCH aplica modificações parciais a um recurso.

O método HTTP `PUT` permite apenas substituições completas de um documento. Em contraste ao `PUT`, o método `PATCH` não é idempotente, ou seja, requisições sucessivas idênticas *podem* obter efeitos distintos. Todavia, é possível realizar requisições `PATCH` de modo a serem idempotentes.

`PATCH` (assim como `PUT`) podem ter efeitos colaterais em outros recursos.

Uso do `.partial` no ZOD

`.partial`

Inspired by the built-in TypeScript utility type [Partial](#), the `.partial` method makes all properties optional.

Starting from this object:

```
const user = z.object({
  email: z.string(),
  username: z.string(),
});
// { email: string; username: string }
```

ts

We can create a partial version:

```
const partialUser = user.partial();
// { email?: string | undefined; username?: string | undefined }
```

ts

Logging

Use o `PrismaClient` `log` parâmetro para configurar [níveis de log](#), incluindo avisos, erros e informações sobre as consultas enviadas ao banco de dados.

O Prisma Client suporta dois tipos de registro:

- Fazendo login em [saída padrão ↗](#) (padrão)
- Registro baseado em eventos (use `$on()` o método para [assinar eventos](#))

Log to stdout

The simplest way to print *all* log levels to stdout is to pass in an array `LogLevel` objects:

```
const prisma = new PrismaClient({
  log: ['query', 'info', 'warn', 'error'],
})
```

Relacionamentos entre tabelas do B.D.

É a associação entre as informações de diferentes tabelas.

São estabelecidos por meio de chaves, que são atributos (colunas) que identificam os registros de cada tabela.

Os relacionamentos entre tabelas permitem:

- Evitar dados redundantes
- Consultar dados de forma eficiente
- Criar relatórios
- Organizar e recuperar os dados de forma eficiente

Relações

Uma relação é uma conexão entre dois modelos no esquema Prisma. Por exemplo, há uma relação um-para-muitos entre `User` e `Post` porque um usuário pode ter muitas postagens de blog.

O esquema Prisma a seguir define uma relação um-para-muitos entre os modelos `User` e `Post`. Os campos envolvidos na definição da relação são destacados:

Bancos de dados relacionais MongoDB

```
model User {
    id      Int      @id @default(autoincrement())
    posts Post[]
}

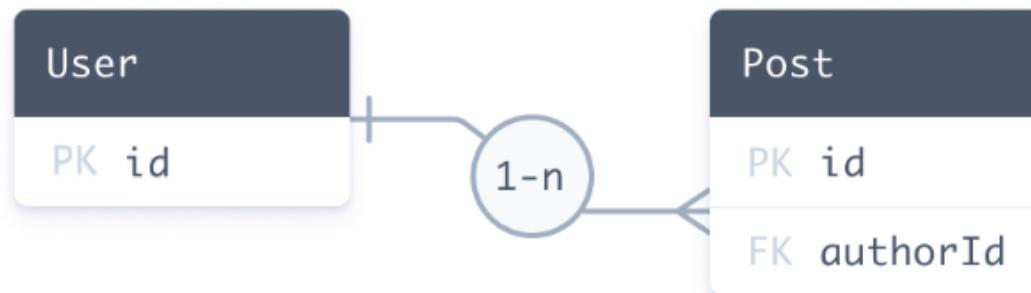
model Post {
    id      Int      @id @default(autoincrement())
    author User @relation(fields: [authorId], references: [id])
    authorId Int // relation scalar field (used in the `@relation` attribute above)
}
```

No nível Prisma ORM, a relação `User` / `Post` é composta de:

- Dois [campos de relação](#) : `author` e `posts` . Os campos de relação definem conexões entre modelos no nível Prisma ORM e **não existem no banco de dados** . Esses campos são usados para gerar o Prisma Client.
- O campo escalar `authorId` , que é referenciado pelo `@relation` atributo. Este campo **existe no banco de dados** - é a chave estrangeira que conecta `Post` e `User` .

No nível do Prisma ORM, uma conexão entre dois modelos é **sempre** representada por um [campo de relação](#) em **cada lado** da relação.

O diagrama de relacionamento de entidade a seguir define a mesma relação um-para-muitos entre as tabelas `User` e `Post` em um **banco de dados relacional**:



Em SQL, você usa uma *chave estrangeira* para criar uma relação entre duas tabelas. Chaves estrangeiras são armazenadas em **um lado** da relação. Nossa exemplo é composto de:

- Uma coluna de chave estrangeira na `Post` tabela chamada `authorId`.
- Uma coluna de chave primária na `User` tabela chamada `id`. A `authorId` coluna na `Post` tabela faz referência à `id` coluna na `User` tabela.

No esquema Prisma, o relacionamento chave estrangeira/chave primária é representado pelo `@relation` atributo no `author` campo:

```
author    User    @relation(fields: [authorId], references: [id])
```

Relacionamentos: Exemplo

```
model Marca {  
    id          Int      @id @default(autoincrement())  
    nome        String   @db.VarChar(30)  
    cidade      String   @db.VarChar(30)  
    representante String  @db.VarChar(40)  
    fone        String   @db.VarChar(30)  
    vinhos      Vinho[]  
    @@map("marcas")  
}
```

```
model Vinho {  
    id          Int      @id @default(autoincrement())  
    tipo        String   @db.VarChar(30)  
    preco       Decimal  @db.Decimal(9,2)  
    quant       Int      @db.SmallInt @default(1)  
    teor        Decimal  @db.Decimal(6,2)  
    ano         Int      @db.SmallInt  
    marca       Marca   @relation(fields: [marcaId], references: [id])  
    marcaId    Int  
    @@map("vinhos")  
}
```

Brasileirão 2025

Criar uma nova aplicação, o banco de dados (brasileirao_2025_nome_aluno) e as models para definir as tabelas e relacionamentos para cadastrar clubes e jogadores do campeonato brasileiro.

Em **clube** salvar informações como id, clube, estado (criar esquema de validação para que o nome do clube tenha, no mínimo, 3 caracteres e o estado tenha, exatos, 2 caracteres).

Em **jogador** salvar informações como id, nome do jogador, data de nascimento, salario, nacionalidade e o clube a qual ele pertence (criar esquema de validação dos dados).

Realizar o CRUD nas 2 tabelas. Nos clubes, na consulta, exibir também os dados dos jogadores. Em jogadores, exibir o nome do clube.

Na inclusão do jogador, receber a data de nascimento e "manipular" para que ela seja do tipo datetime.

Na listagem do jogador, exibir um campo "manipulado" com a idade do jogador.

Modificar a model jogador para acrescentar o atributo posicao (posição em que o jogador atua – que deve possuir o valor default ""). Rodar a *migration* para atualizar a tabela do banco.

Criar uma rota PATCH para receber id (parâmetro) e novo salário (body) de um jogador.